Intro to Computational Complexity Theory

Suhas Arehalli

COMP221 - Spring 2024

1 Intro to Computational Complexity

- So far, we've been worried about how fast particular algorithms are.
  - But often, we want to know how fast *any* solution to that problem can be.
  - i.e., "Could I write a faster algorithm to solve this problem?"
- In general, it's hard to prove that no faster solution is possible!
  - These are often trivial or near-impossible!
  - i.e., search for unsorted arrays can't be faster than linear time, since we must at least check each index.
  - A more sophisticated argument of this kind is the problem from HW2 where you proved *all* comparison-based sorts are $\Omega(n \log n)$!
- Instead, we'll adopt a *relative* strategy; we will prove a particular problem is at least as hard as some other canonical problem that's known to be difficult!

1.1 (Turing/Cook) Reductions

- Consider the following scenario:
  - We have some problem specification X that we want to prove is hard (i.e., doesn't have a fast solution).
  - Assume we also have some problem specification Y that we *know* is hard — i.e., we know *every* solution to Problem Y is in $\Omega(h(n))$.
  - Let ProbX be an arbitrary algorithm that solves problem X that runs in worst-case time complexity $\Theta(f(n))$
  - Now consider a function Translate that turns a particular *solution* of problem X into a solution for problem Y in time $\Theta(g(n))$.
  - We can then construct a solution to Problem Y in terms of ProbX:

    function ProbY(...)
    $x \leftarrow$ ProbX(...)
    return Translate(x)

end function

- Consider the time complexity of this algorithm: $\Theta(f(n) + g(n))$!
- Since this is a solution to Problem Y, and we know Problem Y is hard, we know this solution is $\Omega(h(n))$.
- Thus we know that $f(n) + g(n) \in h(n)$!
- In the best case, $g(n)$ is dominated by $f(n)/h(n)$, and we can conclude that $f(n) \in \Omega(h(n))$!

- Let's sit with that for a second: If we find a sufficiently fast translation from a solution to Problem X to a solution to Problem Y, **and** we know Problem Y is sufficiently hard, then we know that Problem X is also hard!
  - Hard, so far, means that any algorithm that solves the problem is slower than some bound. We'll hone in on what that bound is in a bit!
- Note that I've elided translation of the *inputs* of the problem — this is actually perhaps the *most* interesting part of the translation process, but bear with me for now!
- For now, we're interested in writing functions like ProbY that use calls to algorithms that solve Problem X (i.e., ProbX) in order to solve Problem Y.
  - This is called a *reduction* of Problem Y to Problem X.
  - Note the ordering here! It's tricky!
    - *Reducing* Problem Y to Problem X means *translating* a solution (or multiple solutions) to an instance of Problem X into a solution to Problem Y.
    - A fast reduction of Problem Y to Problem X proves that Problem X is at least as hard as Problem Y.
  - More technically, the general class of reductions we're considering right now are called *Turing* or *Cook Reductions*. We'll look at a different, more specific kind of reduction later.

1.2 Decision Problems

- In general, it's messy to trying to work with the various kinds of problem specifications we've seen so far.
- In the future, we will discuss complexity theory in terms of *Decision Problems* — problems with yes/no (i.e., true or false) answers.
- We do this because of these kinds of problems are well studied and have a lot of theory around them (i.e., from automata/formal language theory!).
  - To fully understand what's going on here, you should take Theory of Computation!
  - For now, you'll see this pay dividends when we talk about Karp/Many-One reductions, and in general when these problems are just easier to work with.

- The claim underlying this choice is that we can convert interesting algorithmic problem specifications into equally interesting (i.e., just as hard!) decision problem variants!
- To see this, consider the case of *Vertex Cover*:
  - A *Vertex Cover* for a Graph $G = (V, E)$ is a set $C \subseteq V$ such that $\forall e \in E$, some vertex $v \in C$ is incident to e. That is, $\forall (v, w) \in E$, $v \in C$ or $w \in C$ (or both!).
  - **(Minimal) Vertex Cover**: Given a Graph $G = (V, E)$, find the *minimal* vertex cover $C \subseteq V$.
  - **Vertex Cover (Decision)**: Given a Graph $G = (V, E)$ and $k \in \mathbb{Z}_{\geq 0}$, determine whether there is a vertex cover C with $|C| \leq k$
- My claim: We can reduce Minimal Vertex Cover to it's decision variant and vice-versa.
- Consider the following reduction:

```
function VertexCover_Decision(G = (V, E), k)
    C ← VertexCover(G)
    if |C| ≤ k then
        return true
    end if
    return false
end function
```

- This reduces the decision problem to the optimization problem with a constant time transformation of the output!
  - This means that if VertexCover runs in $\Theta(f(|V|))$, then VertexCover_Decision can be solved in $\Theta(f(|V|))$ as well!
  - That is, we showed that VertexCover is at least as hard as VertexCover_Decision!
- Of course, this is not surprising: Most of you probably expected that the decision variant was easier than the optimization version!
- So let's consider the trickier reduction:[1]

```
function VertexCover(G = (V, E))
    k ← 0
    while ¬VertexCover_Decision(G, k) do
        k ← k + 1
    end while
    C ← ∅
    for v ∈ V do
        E' ← E \ {e ∈ E | v incident to e}
        V' ← V \ {v}
        G' ← (V', E')
        if VertexCover_Decision(G', k − 1) then
```

[1] This is different than the algorithm I gave you in class — this one is more straightforward, and doesn't require some of the handwaving I did about modifying the edge set!

```
            C ← C ∪ {v}
            E ← E′
            V ← V′
            k ← k − 1
            if k == 0 then
                return C
            end if
        end if
    end for
end function
```

- To see that this algorithm is correct, observe...
  - ...that the first for-loop will set k to the size of the minimum vertex cover
  - ...that if we enter the if-statement, the minimum vertex cover of size $k-1$ for G' (call it C') covers all edges that v is no incident to!
  - Thus, $C' \cup \{v\}$ is a vertex cover of G of size k!
  - i.e., at each iteration, $C \subseteq C^*$, where C^* is a minimal vertex cover of (initial) G.
  - Write out a formal proof as practice!
- Assume that VertexCover_Decision runs in $\Theta(f(|V|))$ time. Then this algorithm runs in $\Theta(|V|f(|V|))$ time!
- Is this good enough?
  - This is a *weird* reduction — we call the decision problem's algorithm order $|V|$ times!
  - In terms of asymptotic time complexity, this reduction is an order of magnitude slower!
  - I'll still claim that this difference doesn't matter *for the kinds of things we care about*!
- What do we care about? *Tractability*!
- A problem X is *tractable* if there exists an algorithm that solves that problem in polynomial time. That is, for some $c \in \mathbb{Z}$, there exists a $O(n^c)$ algorithm that solves problem X!
  - Note that this means that an algorithm is *intractable* if there exist **no** polynomial time solutions.
  - Proving tractability means finding a polynomial time algorithm to solve X. Proving intractability means proving **all** algorithms that solve problem X run slower than polynomial time!
- Note that, given this second reduction, we can still claim that if VertexCover_Decision is tractable, then VertexCover is tractable!
  - Perhaps more importantly, this reduction tells us that if VertexCover is *in*tractable, then VertexCover_Decision is also intractable!
- With both reductions, we know that, by our tractability-based definition of hardness, both problems are equally hard!
- Thus, moving forward, we'll be working nearly exclusively with decision problems rather than more complex problem specifications.

2 (Karp) Reductions

- Now we will stay squarely in the world of decision problems, and this buys us the possibility of a stronger (and more theoretically satisfying!) kind of reduction: A *Karp*, or *Many-to-one* reduction!
- As promised, this is also where we start thinking about translating from inputs to inputs!
- Let X and Y be problems, ProbX and ProbY algorithms that solve those problems. A *Karp* reduction from Y to X is a transformation Translate such that for any input I to ProbY, ProbY(I) = ProbX(Translate(I)).
  - That is, in algorithmic form, we can write a solution to Problem Y in the form

    function ProbY(I)
    $I' \leftarrow$ Translate(I)
    return ProbX(I')
    end function
- This is a very powerful kind of reduction: we know that Problem Y isn't just solvable using Problem X (like in a Turing Reduction), but that instances of Problem Y are *special cases* of Problem X!
  - Translate then tells us exactly which instances of Problem X correspond to each instance of Problem Y.
- Two things are critical here: Because we're translating problem instances/inputs,
  - ... we can only call an algorithm that solves problem X exactly once!
  - ... we can't transform the *output* of the algorithm in any way!
- The first point is a big restriction, but the second is subtly important.
  - Constructing these kinds of reductions is only really possible if we work with decision problems, because then we just need both algorithms to say true or false!
  - This also means that even within decision problems, we can't do things like negate outputs — We have to construct a mapping such that instance I of problem X is true iff some instance I' of problem Y is true!
- From now forward, when we discuss reductions for decision problems, I'm going to be focusing on Karp reductions.
  - If any of you are interested in Theoretical CS, this is the kind of reduction you'll see by default (Skiena doesn't even draw this distinction).

3 Canonically Hard Problems

- Remember our goal here is to convince ourselves some problems are hard.
  - We now have a theory that helps us say Problem X is at least as hard as Problem Y (i.e., we can reduce Problem Y to Problem X).

  - To show some Problem X is hard, we now need to find sufficiently hard problems Y that we can reduce to X!
  - Remember our formalization of the idea of a problem being hard is a problem being *intractable*.

- It follows that our goal should be to find intractable problems to reduce to problems we think are hard!

- If we want to be really careful about the term intractable, we're kind of out of luck...
  - What if we're just not clever enough to find a polynomial time solution?

- Instead, we will "settle" for problems that we're very confident are intractable: problems that are *NP-hard*

- A problem in complexity class *NP* (Non-deterministic Polynomial time) is able to be *verified true* in polynomial time.
  - A *verifier* is an algorithm that, for a particular problem, takes in a problem instance and a *certificate/witness* — some information that serves as evidence that the purported answer is correct — and concludes whether the witness justifies a true or false answer.
  - There is some subtlety here: What we want is that there *exists* some certificate such that our verifier will confirm that the right answer is correct in polynomial time.
  - This witness essentially acts as a hint to our algorithm: Is there some information that I can give you such that you can solve the problem in polynomial time?
    * In practice, that hint is usually in the form of the solution to the non-decision version of the problem: A vertex cover, or a variable assignment, or Hamiltonian path, etc.
  - For instance, consider the HamiltonianCycle decision problem. A *certificate* would be the sequence of edges in the graph, and a *verifier* for true answers would be an algorithm like

```
function HP_Verifier(G = (V, E), C)
    (v, w) ← C[1]
    curr ← v
    for (v, w) ∈ C do
        if (v, w) ∉ E or v ≠ curr or w is visited then
            return false
        end if
        curr ← w
    end for
    for v ∈ V do
        if v not visited then
            return false
        end if
    end for
    return true
end function
```

  - For any case where the Hamiltonian Path problem's answer is true, we can *verify* that answer by providing a certificate: a simple cycle in G that visits every vertex!
  - For those who want to go above and beyond: There is a complementary class of problems called $coNP$, which is the set of problems that can be verified *false* in polynomial time. Whether $NP = coNP$ is an open question!
  - The relationship between this verification definition and the *nondeterministic* in NP comes from automata theory. You can think about nondeterminism as a sort of parallelism: If you could try to verify evert possible certificate at once, you can solve the problem in polynomial time!

- It should be intuitive that $P \subseteq NP$: If you can solve it from scratch in polynomial time, you can verify the problem in polynomial time.
- However, we're not certain polynomial time solutions aren't possible for problems in NP!
  - Literally a million dollar problem! See the Millenium Prize Problems!
- But, given all the attempts at solving these problems, and the fact that a large group of these problems have be reduced to each other (NP-*Complete* problems), they are hard enough for our purposes
  - *Functionally* intractable!

3.1 SAT/3-SAT

First, some terms:

- A *variable* v takes boolean values
- A *literal* is either a variable v or it's negation $\neg v$
- A *clause* is a chain of disjunctions (logical ors) of literals: $v_1 \vee v_2 \vee \neg v_3$

Then we get *SAT* and *3-SAT*!

SAT
For: A set of variables V and a set C of clauses constructed from those variables.
Determine: Whether there exist a set of assignments of true or false to each variable such that each of the clauses evaluates to true.

3-SAT
For: A set of variables V and a set C of clauses of *3 literals* constructed from those variables.
Determine: Whether there exist a set of assignments of true or false to each variable such that each of the clauses evaluates to true.

3.2 Hamiltonian Cycle/Path

The unweighted version of the traveling salesman problem

Hamiltonian Cycle/Path
For: A Graph $G = (V, E)$
Determine: Whether there exists a *simple* cycle/path that visits every vertex in V once.

3.3 Vertex Cover/ Independent Set

Vertex Cover and Vertex Cover's counterpart!

A *vertex cover* for an (undirected) graph $G = (V, E)$ is a set $C \subseteq V$ such that for every $(v, w) in E$, either $v \in C$ or $w \in C$.

An *independent set* for an (undirected) graph $G = (V, E)$ is a set $S \subseteq V$ such that for any $v, w \in S$, $(v, w) \notin E$

Vertex Cover
For: A Graph $G = (V, E)$ and $k \in \mathbb{Z}_{\geq 0}$
Determine: Whether there exists an vertex cover C with $|C| \leq k$.

Independent Set
For: A Graph $G = (V, E)$ and $k \in \mathbb{Z}_{\geq 0}$
Determine: Whether there exists an independent set S with $|S| \geq k$.

3.4 Integer Partition

A *multi-set* is just a set that allows duplicates. That is, an un-ordered collection with repetitions.

Integer Partition
For: A multi-set of integers S
Determine: If there exists some partition S_1, S_2 of S such that $\sum_{s \in S_1} s = \sum_{s \in S_2} s$

3.5 And others

It's probably useful to be familiar with other, related NP-Complete problems: Clique shows up often, or something like Linear Integer Programming is probably of practical interest. But the few here (Vertex Cover/Independent Set, SAT/3-SAT, Hamiltonian Cycle/Path, and Integer Partition) cover a few general classes of problems: vertex selection in graphs (Vertex Cover/Independent Set), edge selection in graphs (Hamiltonian Cycle/Path), logic problems (SAT/3-SAT), and combinatorics/counting problems (Integer Partition). As Skiena suggests, and at least for this class, these will cover our bases!